Xpnet: A Graphical Interface to Proof Nets with an Efficient Proof Checker

Jawahar Chirimar   Carl A. Gunter   Myra VanInwegen[1]

Department of Computer and Information Science
University of Pennsylvania, Philadelphia, PA 19104

We describe an automated proof-checker for linear logic proof nets based on efficient algorithms for deciding necessary and sufficient *path conditions*. The system, which is called *Xpnet*, is an experiment in the graphical representation of mathematical proof objects.

Linear logic is a new logical system introduced by Jean-Yves Girard [Gir87]. The main difference between classical and linear logic is that in the latter structural rules for copying or deleting assumptions are disallowed. Full linear logic is ordinarily presented as a sequent calculus in the style of Gentzen. Proof nets, which are the linear logic analogs of natural deduction, can be formed from a subset of the logic known as the *multiplicative fragment*.

Proof nets are represented as graphs with two types of edges (see Figure 1). Their most distinctive feature is the condition required for *proof-checking*. In most logics proof-checking involves verifying that each step of a proposed proof object is an instance of an axiom or rule of the system. This is insufficient for proof nets: checking the steps by which the graph is constructed only ensures that it is a *proof structure*, a type of graph that is a proof net exactly when it satisfies a special connectivity property. Stated mathematically, this property sounds as if it would take exponential time to check. Girard asserted that his path condition for proof nets could be checked in time proportional to a polynomial in the number of edges. We have developed an algorithm with this efficiency using an equivalent condition given by Danos and Regnier [DR89].

Checking even moderately-sized proof structures by hand can be very tedious. The motivation behind Xpnet (for X Proof NET) was to use an implementation of our algorithm to automate this process. Since proof nets can be described pictorially it is natural to have a graphical user interface to aid in the sketching of proof structures to be checked.

There are relatively few systems for the graphical manipulation of mathematical objects that can be said to 'understand' the objects they manipulate. An example of a system that does is LatticeDraw, designed for the MacIntosh by Alan Day. This program allows the user to construct an ordered collection of elements (a lattice) by drawing its Hasse diagram. It aids in the creation of a well-formed Hasse diagram by guiding legitimate uses of a mouse to describe elements and relationships between them. It is then capable of providing information about the lattice by responding to questions selected from a menu such as 'show me all of the primes'. It is also capable of calculating and displaying related lattices. For example, the program can calculate the lattice of lower sets of a lattice and display its Hasse diagram. This

---

[1] The authors' email addresses are `chirimar`, `gunter`, and `myra @saul.cis.upenn.edu`. Gunter's research is partially supported by NSF grant CCR-8912778 and an ONR Young Investigator award. Chirimar's research is supported by NSF grant CCR-8912778. VanInwegen's work is supported by ARO grant DAALO3-89-C-0031.

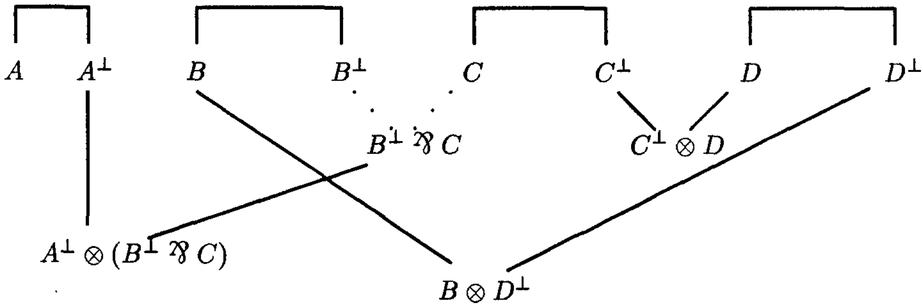

Figure 1: A proof net

might take hours of work to do by hand, even for fairly small examples.

Other examples of systems with a graphical interface to a mathematical system are the programs Turing's World and Tarski's World of Jon Barwise. The first of these aids the user in providing a mouse sketch of a Turing machine which can then be run as a program. The latter determines the truth of logical propositions about a configuration of blocks which the user can create with a mouse.

The 'look and feel' of Xpnet is partly inspired by LatticeDraw. Axioms and rules are placed by mouse clicks, and line displays guide the user in the legitimate placement of his nodes. Diagnostics are carried out by clicking on a button. Unlike LatticeDraw, Xpnet is implemented using the X Toolkit and runs under the X Window System. Its internal structure is similar to that of Xfig. The system is portable and robust as well as easy and fun to use.

Below we describe how to use Xpnet to build and manipulate proof structures, and then we discuss the basis and implementation of our proof-checking algorithm.

Description of the Program. Figure 2 shows Xpnet in operation. There are three main windows in its display area. The canvas is the large window in which proof structures are drawn. The message window is below the canvas; messages to the user are printed there. The panel (for control panel) is to the left of the canvas; users initiate commands by clicking on the buttons it contains.

A proof structure is built of components called *clusters*. The cluster types are axiom, pax (short for proper axiom), tensor, par, and cut, and they are placed in the structure using command buttons in the panel. Xpnet employs user-selected ASCII strings to represent the symbols for tensor ($\otimes$), par (⅋), and the linear negation perp ($^\perp$); here we use the defaults “tensor”, “par”, and “~”.

To build a proof structure one starts off with axiom and pax clusters. Axiom clusters have as formulae A and $A^\perp$, where A is an atomic formula entered by the user. Pax clusters may have any formulae; the user types in the name of a file that contains them. In both cases, the user selects the position of the formulae with the mouse and then determines the height of the solid edge connecting the formulae.

After placing axioms or paxes the user can add tensor, par, or cut clusters. Each requires the selection of two premises. A formula may be used at most once as a premise. To place a tensor or par cluster, the user selects the premises (by clicking on the formulae) and then places the resulting formula. The position of the formula

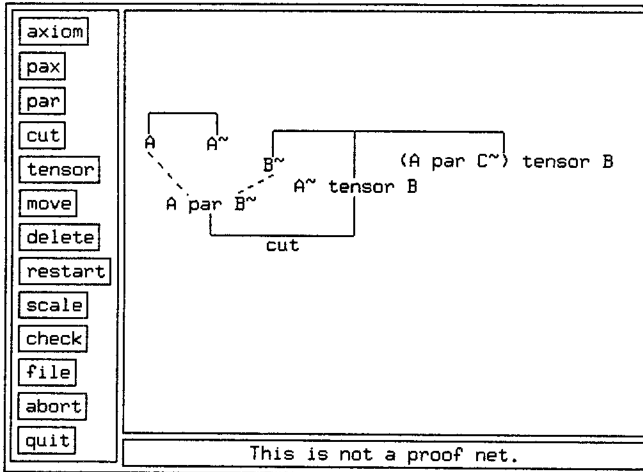

Figure 2: Xpnet with a proof structure that is not a proof net.

is forced by the program to be below the premises. Edges (solid for a tensor, dotted for a par) connect the formula to its premises. Since these formulae can grow to be rather large, the user has the option of displaying only the type of the cluster. The premises for a cut cluster must be inverses. After selecting the premises for a cut, the user selects the position of the solid edge below the premises which connects them. No formula is generated by the cut cluster. If the resulting proof structure is a proof net, then the conclusions of the proof are the formulae that are not used as premises. For example, the conclusions of Figure 1 are A, $A^{\perp} \otimes (B^{\perp} \parr C)$, $B \otimes D^{\perp}$, and $C^{\perp} \otimes D$.

Examples of axiom, pax, par, and cut clusters appear in Figure 2. The axiom consists of the nodes labeled `A` and `A~` connected above with a bar. The pax consists of the nodes labeled `B~`, `A~ tensor B`, and `(A par C~) tensor B` connected above with a bar. The premises of the par are `A` and `B~`, and the resulting formula is `A par B~`. The premises of the cut are `A par B~` and `A~ tensor B`, which are connected below with a bar. Figure 1 contains three tensors.

Editing is supported by the *move, delete, restart,* and *scale* commands. The *move* command moves a connected component of the graph. The *delete* command deletes a cluster if none of its formulae are used as premises. The entire proof structure can be erased using *restart* or scaled by an arbitrary amount using *scale.*

Once a proof structure has been drawn, the user can obtain information about it using the *expand* and *check* commands. The *expand* command causes the entire formula corresponding to a node to be printed out in the message window and is only useful when the user chooses to have only the cluster type displayed at tensor and par clusters. The *check* command invokes the proof net verification algorithm, and the result (`Net is ok.` or `This is not a proof net.`) is printed to the message window. In Figure 2, the *check* command has just been applied.

The *file* command brings up a menu with several options on it. The *save* and *read* options allow the user to save a textual representation of a proof net to a file

and to retrieve one of these files. The *xpn2ps* option dumps the proof structure in a file in PostScript format.

The *abort* command terminates a command already in progress. The *quit* command terminates the program.

Proof-Checking. A naive implementation of proof-checking based on the condition for proof structures to be proof nets takes exponential time. We define a *switching* of a par cluster to be the solidification of one of the edges from the formula to its premises and the deletion of the other. A switching of a proof structure is the selection of a switching for each par in the graph. The Danos and Regnier condition for a proof structure to be a proof net is that each graph resulting from a switching of the proof structure is a tree. As there are two possible switchings for each par, checking this directly takes time exponential in the number of pars in the structure.

The idea for our proof-checker comes from the *Sequentialization Theorem*, which states that for any proof net there is a sequent proof with the same conclusions. The proof of this theorem provides a method by which a sequent proof is constructed from a proof net. This method, which is shown to be correct using the Danos and Regnier condition, can be used to determine whether or not a given proof structure is a proof net: we check if it is possible to construct an equivalent sequent proof, and if so, the structure is a proof net. Structures consisting of just an axiom or pax cluster are the base case. Converting these to sequent proofs is trivial, so such structures are proof nets. To convert a structure with a par conclusion we remove the par, convert the rest of the structure, and then add a par rule to the resulting sequent proof. To check such a structure, we remove the par cluster and check the rest of the structure. The only cases left are structures with only tensor conclusions and cuts, and structures consisting of two or more axioms or paxes. The latter are not connected and so are not proof nets.

The difficulty is in structures that have only tensor conclusions and cuts. To convert these we would like to remove a tensor or cut, convert the two resulting structures, and apply the appropriate rule to the conversions. However, removing an arbitrary cluster may not split the proof structure into two separate pieces: in Figure 1 we can split on only the leftmost tensor. The *Tensor Splitting Lemma* states that we can split on at least one cluster in a proof net. Thus to check such structures, we search for a cluster to split on, at the same time checking that the graph is connected. If we succeed, then we split and check the two resulting structures. To determine if we can split on a cluster D with premises A and B, we first mark D with the number 3 and all the other formulae in the structure with 0. Then we mark with 1 both A and all formulae connected to it by graph edges except the one between A and D. If in doing so we encounter D, then we have come to D through B, so removing D will not split the structure. If we succeed, we mark with 2 both B and all formulae connected to it by edges except the one between B and D. If there are any formulae still marked 0, then the structure is not connected.

The algorithm above is order of E^3, where E is the number of edges in the graph. To check a structure we remove a cluster and then verify what is left. Removing an axiom, pax, or par takes time proportional to the number of nodes in the cluster. Removing a tensor or cut takes $O(NE)$, where N is the number of nodes: we test $O(N)$ clusters to find one to split on, and this test requires $O(E)$ time. Thus checking

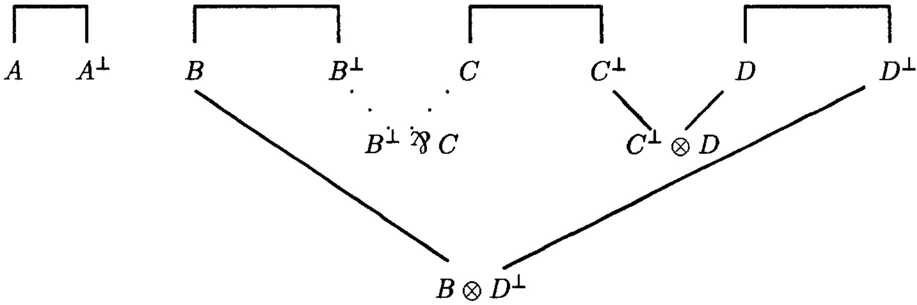

Figure 3: Verification: after splitting

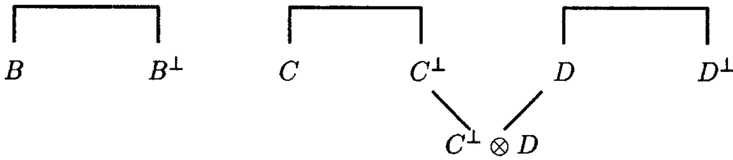

Figure 4: Verification: after removing par and splitting again

the entire graph takes $O(N^2E)$ time. Since in a proof structure the number of edges is proportional to the number of nodes, this is $O(E^3)$.

To demonstrate the algorithm, we will go through the steps involved in checking the structure in Figure 1. After splitting on the leftmost tensor, we get the proof structure in Figure 3. The structure on the left is immediately a proof net since it is an axiom. To check the other one, we remove the par and can then split on either tensor. Say the bottom one is chosen, resulting in Figure 4. Again the left hand structure is immediately a proof net, and we split the right one on the remaining tensor, getting two structures which are axioms.

Obtaining the program. To get an executable that runs on the Sun4, FTP to ftp.cis.upenn.edu and log in as anonymous (give your email address as password). The file is in the pub directory and is called xpnet.tar.Z. To use it you must uncompress it and extract the contents from the resulting tar file. One way to do this is "`uncompress < xpnet.tar.Z | tar -xf -`" which will place the extracted files in the current directory.

We would like to acknowledge assistance and encouragement from Jean Gallier, Vijay Gehlot, Andre Scedrov, Hay Tran, and Loan Dinh in our efforts on this project.

References

[DR89] V. Danos and L. Regnier. The structure of multiplicatives. *Archive for Mathematical Logic*, 28:181–203, 1989.

[Gir87] J. Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.